CG2111A Engineering Principle and Practice II
Semester 2 2024/2025

"Alex to the Rescue"
Final Report
Team: B02-4B

Preface

Engineering is rarely a straight line, it's more often a winding path of late nights, head-scratching bugs, improbable fixes, and the occasional "aha!" moment that feels like magic. This report is not just a summary of our technical work, but a record of our journey: a tale of problems faced head-on, systems reimagined, and a robot named Alex who sometimes had a mind of his own.

From the first hesitant movements of a default build to a fully functioning system that could navigate, detect, grip, and react, we walked the fine line between precision and chaos, theory and tinkering, frustration and delight. Along the way, we discovered that elegance in design is not just an aesthetic pursuit, but often the key to reliability and success.

This project could not have been done without the assistance of those around us. We appreciate the professors and the teaching team of CG2111A for imparting invaluable knowledge and insights, guiding us at every step, especially Prof Henry Tan, our kind and exciting professor and the extremely helpful staff at the DSA lab.

We invite you to read not just with a technical lens, but with curiosity. Because at the heart of every line of code and every servo twitch is a team of students who dared to turn ideas into movement, and had fun along the way.

With this, we proudly present our blood, sweat and tears, our Alex.

- Members of team B02-4B

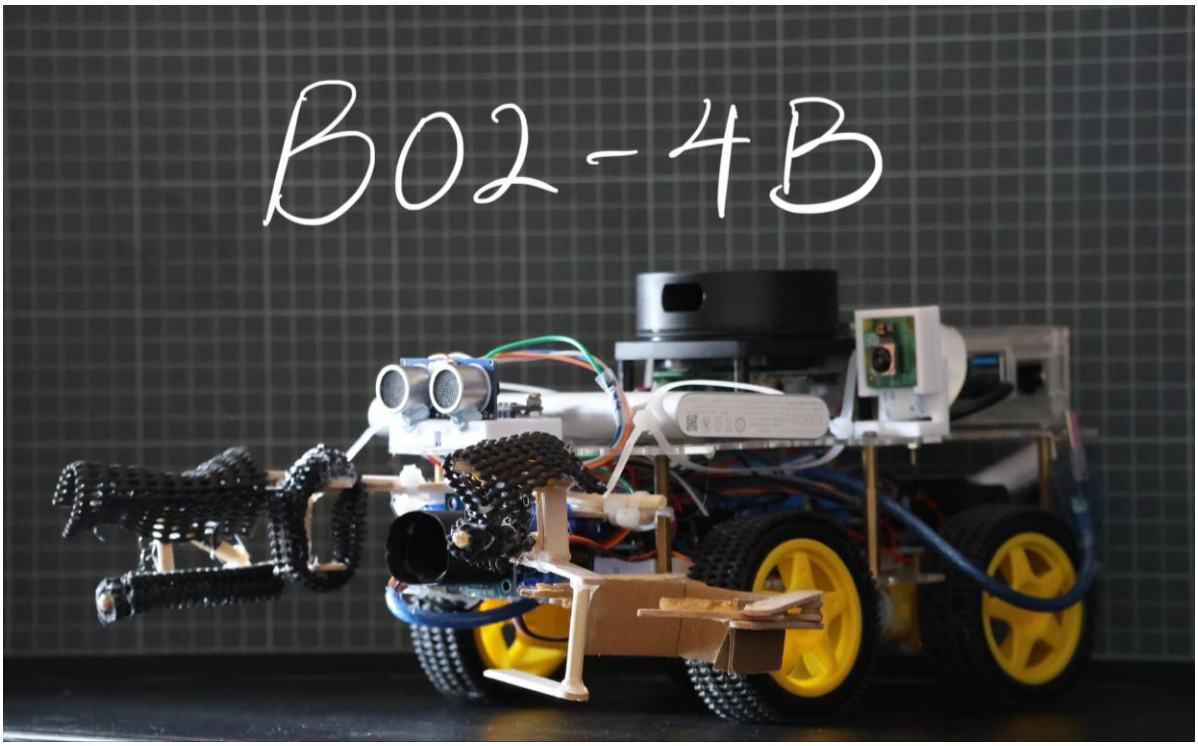


Table of Contents

BNO085 Implementation to Retrieve Real-time Yaw Angles (Alex's headings)

```
#include <Adafruit_BNO08x.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>
#include <math.h>

#define BNO08x_SDA 20
#define BNO08x_SCL 21
#define BNO08x_ADDRESS 0x4B // From I2C scan, I2C Default Address is 0x4A
Adafruit_BNO08x bno = Adafruit_BNO08x(-1); // -1 means no reset pin
float currYaw = 0.0;
float yawScaleFactor = 1.11273;
float yawOffSet = 20.07273;

void IMUInit() {
    // scanI2C();
    Wire.begin();
    if (!bno.begin_I2C(BNO08x_ADDRESS, &Wire)) {
        while (1);
    }
    // Set the sensor to use the SH-2 sensor fusion algorithm
    bno.enableReport(SH2_ROTATION_VECTOR); // yaw
}

/**
 * Get BNO08x Yaw readings from quaternion data using sh-2 sensor fusion
 */
sh2_SensorValue_t sensorValue;

float getSH2Yaw() {
    // Sensor fusion data structure
    if (bno.getSensorEvent(&sensorValue)) {
        if (sensorValue.sensorId == SH2_ROTATION_VECTOR) {
            // Convert quaternion to Yaw
            float qw = sensorValue.un.rotationVector.real;
            float qx = sensorValue.un.rotationVector.i;
            float qy = sensorValue.un.rotationVector.j;
            float qz = sensorValue.un.rotationVector.k;
            float siny_cosp = 2.0f * (qw * qz + qx * qy);
            float cosy_cosp = 1.0f - 2.0f * (qy * qy + qz * qz);
            float currYaw = atan2(siny_cosp, cosy_cosp) * 180.0f / PI;
            if (currYaw > 180) {
                currYaw -= 360.0;
            } else if (currYaw < -180) {
                currYaw += 360.0;
            }
            return currYaw;
        }
    }
    return currYaw;
}

float getYaw() {
    return yawScaleFactor * getSH2Yaw() + yawOffSet;
}
```

.....26

Section 1 - Introduction

The Moonbase CEG Rescue Mission is a simulated Search and Rescue task accomplished by a robot "Alex". A team of operators will remotely teleoperate Alex from a separate room to complete tasks efficiently, as illustrated using the example below in *Figure 1*.

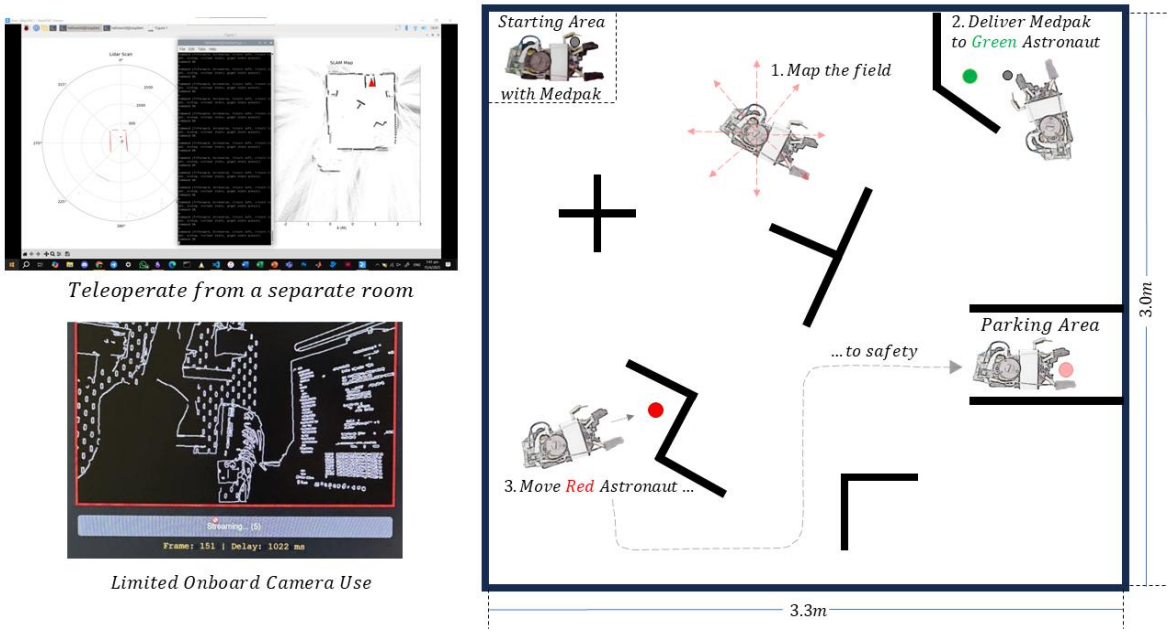


Figure 1: Illustrative Example of Alex's "Search and Rescue" Mission

From the starting point, as Alex maps the unfamiliar environment using its RPLIDAR, operators navigate Alex by using the map generated via SLAM in real-time and by sending commands remotely from their laptop. Alex's main objectives are to correctly identify Green and Red astronauts while avoiding obstacles, and perform rescue operations of delivering a *Medpak* to the Green Astronaut and retrieving the Red Astronaut back to safety using an actuator mechanism designed by the team. Operators may use the onboard Raspberry Pi Cam 4 times for 10 seconds each, but not for navigation purposes. Thereafter, teams will need to submit an accurate hand-drawn map and the team's overall time efficiency will be evaluated. The following sections will describe our team's implementations in detail.

Section 2 - Review of State of the Art

To achieve the functionalities stated in *Section 1* effectively and efficiently, we drew inspiration from cutting-edge teleoperation platforms specifically for Search and Rescue missions, as reviewed next. Their images are included in *Appendix A*.

2.1 PackBot 510

PackBot 510 is a rugged, mobile search-and-rescue robot widely used for military, disaster response, and hazardous environmental operations. It features tracked locomotion to navigate rough terrain, a robotic arm to manipulate objects, and an array of sensors including cameras, LiDAR, infrared, and gas sensors. Its tracks give it the ability to climb obstacles and stairs, and traverse rough terrains like rubble, sand, or snow. The PackBot 510 is compact at 55cm x 85cm and a weight of 27kg, allowing it to fit through narrow spaces. PackBot 510 is highly versatile, being able to switch out its attachments, it is extremely adaptable to fulfill various specific operational requirements. That being said, the PackBot's small size comes with limitations - a limited battery life of 2-4 hours as well as a limited payload capacity.

2.2 Valkyrie

Valkyrie is a humanoid robot developed by NASA to support future human exploration and rescue missions, especially in hazardous environments like space and disaster zones. A wide range of sensors is spread across the entire robot. On its head, stereo cameras, depth cameras, high-definition cameras and LIDAR sensors are mounted to provide clear vision of the surroundings to assist with navigation. Inertial Measurement Units (IMUs) are placed in its torso to help maintain Valkyrie's balance and stability. Valkyrie's hands and arms are equipped with force sensors and tactile sensors to increase precision when interacting with objects. Cameras are also used to provide a better vision of object interaction. Finally, force and pressure sensors are equipped in Valkyrie's legs and feet, providing a better understanding of ground contact and force distribution. The valkyrie has the ability to perform complex human-like tasks like maintenance, construction and even scientific experiments. However, due to its large size and human-like form, the Valkyrie's stability is incomparable to other robots like the PackBot for example. It has limited mobility on rough terrain and is unable to explore confined spaces.

Section 3 - System Architecture

The diagram in *Figure 2* presents an overview of Alex's system architecture. The yellow boxes are hardware components while the blue boxes are software implementations. The red box encapsulates the Master Control Program (MCP) that runs when `alex_main.py` is called, and mimics the PubSub model of ROS implementation, shown in the lower half of *Figure 2*.

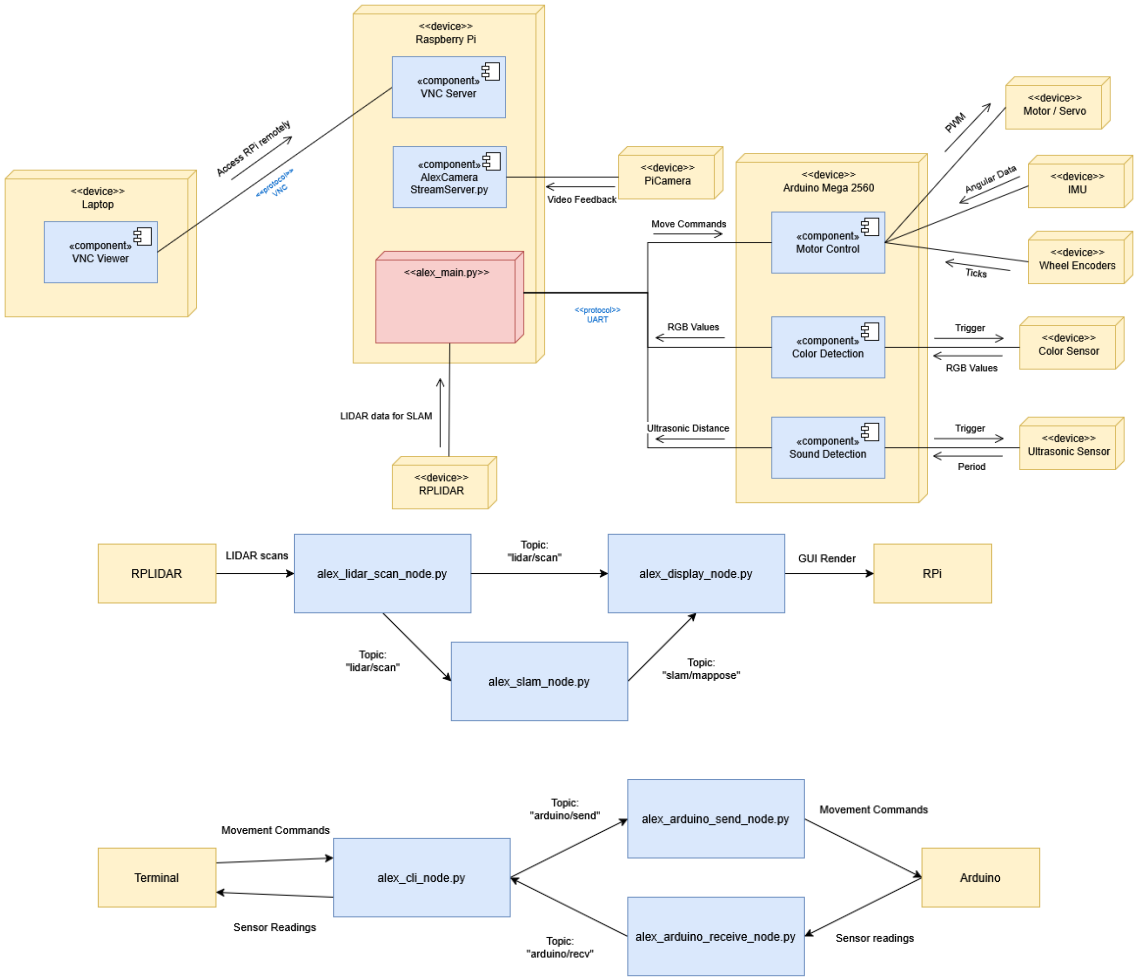


Figure 2: Illustration of the High-Level Components of Alex and the PubSub Model

Our Alex was mainly controlled via the MCP on the single board computer (Raspberry Pi). The RPLIDAR sends its data via serial to the RPi for SLAM to map the terrain. Using Virtual Network Computing (VNC), commands are sent remotely on a laptop to the RPi for controls and the camera (*see Section 6 for details)*. The MCP communicates commands via UART to the Arduino which executes the corresponding codes on the peripheral devices to gather information, traverse through the map, or perform rescue tasks (*see Section 5 for details).*

Section 4 - Hardware Design

This year's mission expanded the potential in innovative mechanical design, especially with the addition of an actuator mechanism required to relocate the Red Astronaut. On top of this, many mechanical considerations were made to optimise our performance, detailed here.

4.1 Alex's Final Form

At first glance, one may be deceived by the clean aesthetics of our robot. Alex was built with space efficiency without the need for any extra acrylic layers. With the two given acrylic mounting plates, chassis standoffs and 4 yellow PMDC motors as its core structure, many features are neatly hidden within its smart hardware design and layout, to be exposed in subsequent segments. Here in *Figure 3*, the hardware components are labelled in detail.

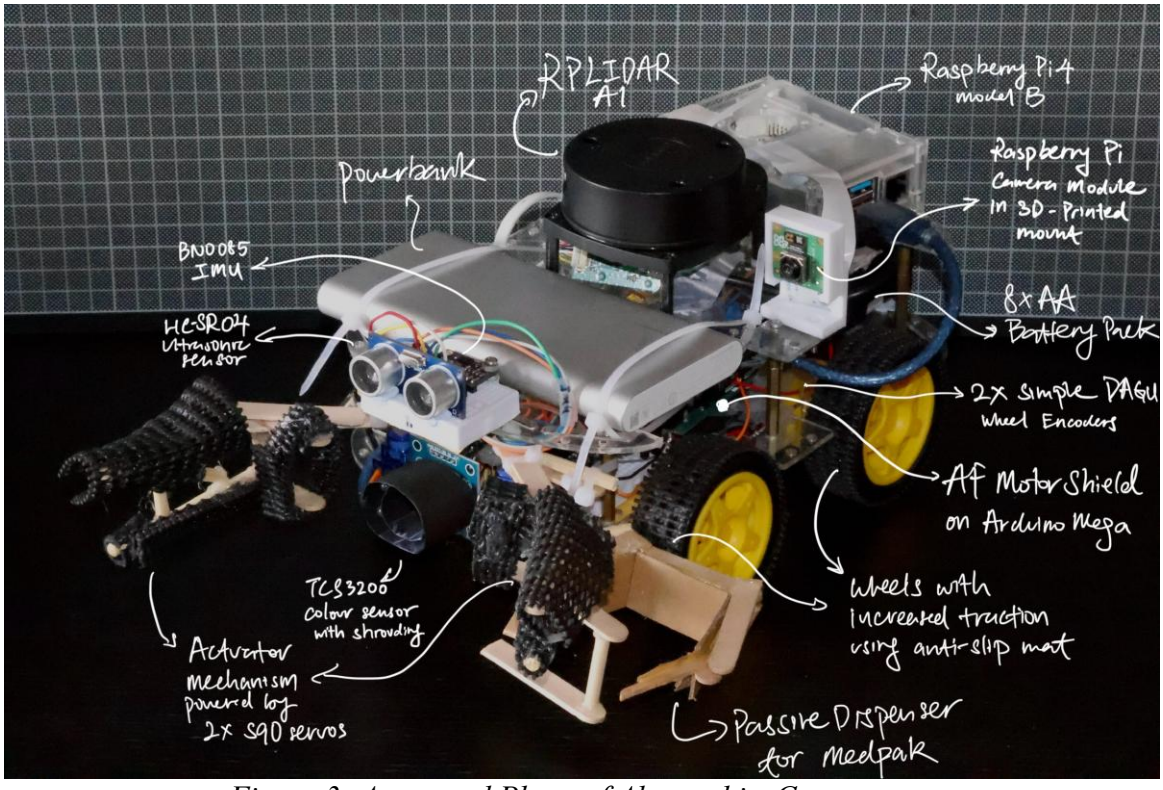


Figure 3: Annotated Photo of Alex and its Components

4.2 Overall Mechanical Design Considerations

As a four-wheel-driven robot, Alex's design prioritises balance and stability by placing its center of gravity (CG), and consequently its center of rotation, roughly equidistant from all four wheels (see *Figure 4*). This configuration ensures that each wheel exerts equal force during movement, minimizing slippage and maximizing efficiency. To achieve this, we strategically distributed weighty components such as the power bank and the 8xAA battery pack across the chassis. We even accounted for the weight of a Red Astronaut figure as part of the final setup.

The RPLIDAR is positioned directly over the center of rotation to minimise translation when turning, enhancing the consistency of SLAM data. This central mass distribution also benefits IMU readings by reducing unwanted tilts and drifts, and enables Alex to pivot smoothly and navigate tighter spaces with ease. To adapt to the dusty, uneven surfaces of our environment, the wheels are equipped with anti-slip mats, improving traction and reducing skidding.

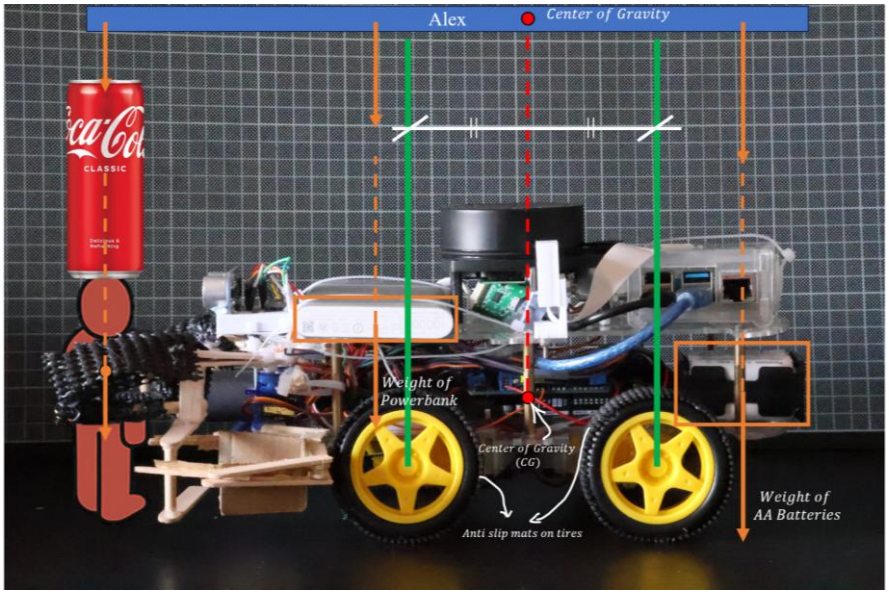


Figure 4: Annotated Side View of Alex, illustrating its Weight Distributions

4.3 Overall Electrical Circuit Design and Schematic

The power architecture is clearly defined:

- 8x 1.2V AA batteries solely for the four PMDC motors through the motor shield
- Clean and stable 5V output from the LiPo power bank for the RPi (and its RPLIDAR, RPi Cam), Arduino Mega, and the peripheral devices (IMU, ultrasonic sensor, colour sensors and servos). The IMU required a workaround, to be explained in *Section 4.4.2*

Electrical connections for the peripheral devices are made via jumper cables and breadboards, except the IMU which requires soldering. Connections are deliberately managed cleanly and securely in the middle section of Alex without any exposed strands. The full schematic diagram made using Altium Designer is available in *Appendix B*.

4.4 Non-Standard Hardware Implementations

During our lab studios, we identified critical problems when running the default Alex build such as its inaccurate and imprecise turnings and poor colour detection. The following are the key solutions we engineered to address those issues and enhance Alex's performance.

4.4.1 BNO085 Inertial Measurement Unit (IMU)

1. High Precision

Precise turnings of ±1° are achieved using yaw angles of high resolution, calculated from BNO085's absolute orientation provided in the form of quaternions using onboard sensor fusion. (*our algorithm to be explained in Section 5.3.2*). Compared to the 8-tick/rev encoders, 1-tick resulted in a roughly 20° turn from our experience, unable to achieve minute turn angles.

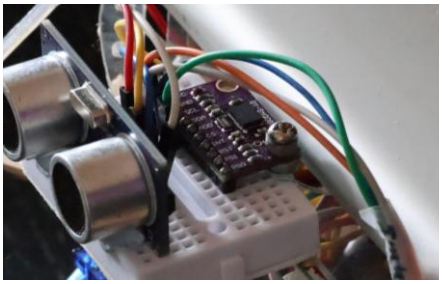


Figure 5: BNO085 IMU

2. High Accuracy and Consistency

The IMU is independent of wheel mechanics, which was a significant weakness in using encoders or time-based movements. Friction between the wheels and floor varies, even with anti-slip material, the degree turned will never be consistent as the wheels skid randomly. However, IMU tracks the robot's headings in real-time at ~100Hz, ensuring that the robot spins until the desired heading is obtained.

4.4.2 Logic Level Shifter and Soldering for the IMU

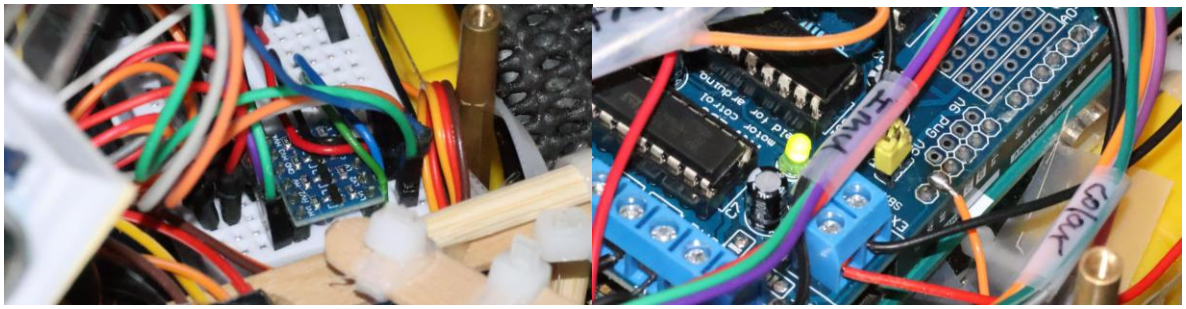
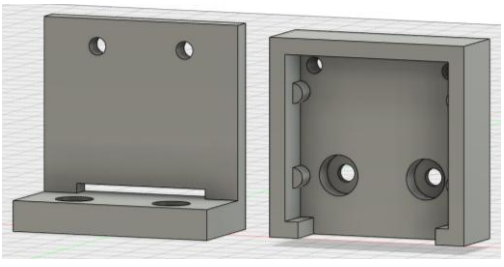


Figure 6: 5V-3.3V Logic Level Shifter (left); Soldered Orange Solid Core to 3.3V Pin (right)

The IMU's maximum operating voltage at supply pin, $V_{DDIO} = 3.63V$ and at any logic pin $V_{non-supply} = V_{DDIO} + 0.3$. This means the BNO085 IMU operates at 3.3V logic levels, and its I2C lines (SCL/SDA) expect 3.3V signals. Since the Arduino Mega sends 5V signals and without a voltage regulator on the IMU's breakout board, a bi-directional 5V-3.3V logic level shifter is needed to step down 5V signals from the Arduino to 3.3 for the BNO085 and vice versa, for the 2-way I2C communications to be reliable. We chose to solder a solid core to the 3.3V pin of the motor shield (*Figure 6, right)* for the most convenient 3.3V source.

4.4.3 3D Printed Mount for Raspberry Pi Camera

The use of the onboard camera is limited and thus precious. We plan to use the camera for attending to the astronauts with precision. Using a rigid mount, our camera is kept firmly secured for optimal viewing angle.



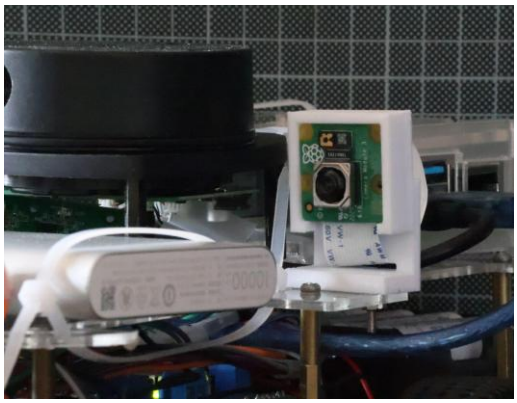


Figure 7: CAD of Camera Mount (left); Mounted Camera on Alex (right)

4.4.4 Black Shrouding for TCS3200 Colour Sensor Module

A double-layered shroud was added to not only reduce the effects of ambient light, but also prevent the sensor's own set of four white LEDs from affecting its readings.

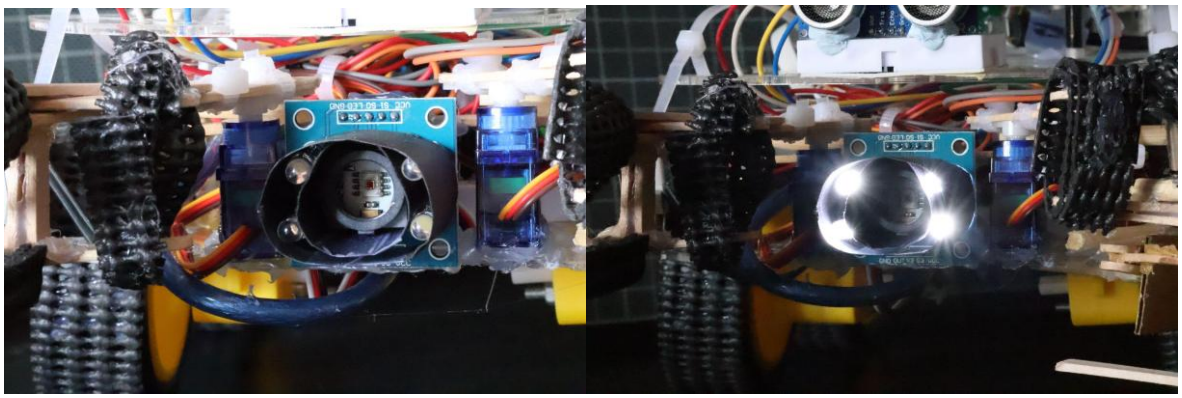


Figure 8: Double-layered Shrouding using Black Paper for Colour Sensor

4.4.5 HC-SR04 Ultrasonic Sensor

The ultrasonic sensor is mounted at the front of Alex to act as its "eyes". This enabled us to consistently achieve the optimal distance from the Red Astronaut before securing it within our actuator, as illustrated in *Figure 9*.

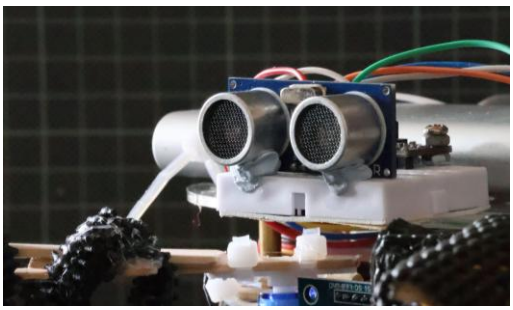


Figure 9: Ultrasonic Sensor (left); Illustration of its Usage (right)

4.5 Elegant Actuator Mechanism Design

"*Elegance is not a dispensable luxury, but a quality that decides between success and failure.*" Edsger Dijkstra's words ignited our engineering minds to design the actuator that is not only effective, but also refined in form, reliable in motion, and mindful of constraints.

4.5.1 Passive *Medpak* Dispenser

Rather than grabbing the *Medpak* or even buying extra servos to dispense it, we engineered a simple and elegant solution, as illustrated in the sequence in *Figure 10*.

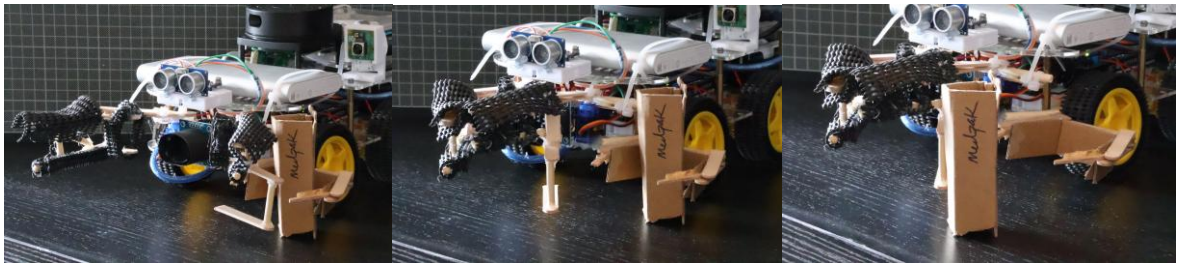


Figure 10: Sequence of Dispensing Medpak, from left to right

Using just 2 ice-cream sticks and cardboard, a storage compartment was built. Extending the right claw, its open-state serves as a "gate" that keeps the Medpak in the storage. (*left picture of Figure 10*). To dispense, we simply close the claw to open the "gate" (*middle picture)* and then reverse Alex (*right picture)*.

4.5.2 Actuator Mechanism

Taking inspiration from structural engineering, we used a truss design for our claws for structural integrity, as illustrated in the left picture of *Figure 11*.

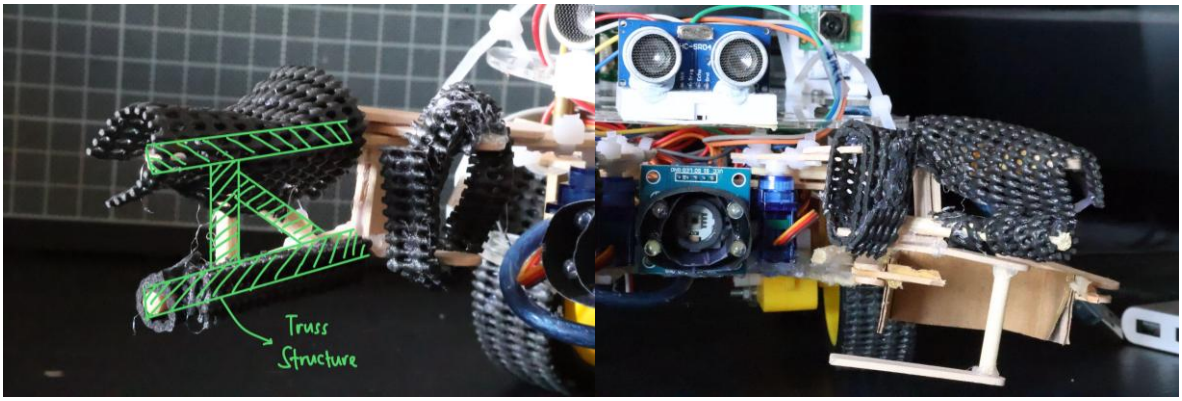


Figure 11: Structure of Claw with Anti-Slip Mat Padding

Since the actual Red Astronauts were inaccessible during preparation, we anticipated the difference in circumference compared to the substitute we used, by using anti-slip mat paddings wrapped around the claw. These paddings not only accommodate variable astronaut sizes, but also give the astronaut a snug fit, so that it does not slip out of our care during transportation. Furthermore, our 4-pronged claw is strategically built to ensure the astronaut's CG lies between the 4 prongs, shown above in *Figure 4*. This structure allows ALEX to securely push and pull the astronaut at high speeds without toppling it over.

Section 5 - Firmware Design

5.1 High Level Algorithm on the Arduino

The following flowchart diagram illustrates the high-level algorithm run on the Arduino.

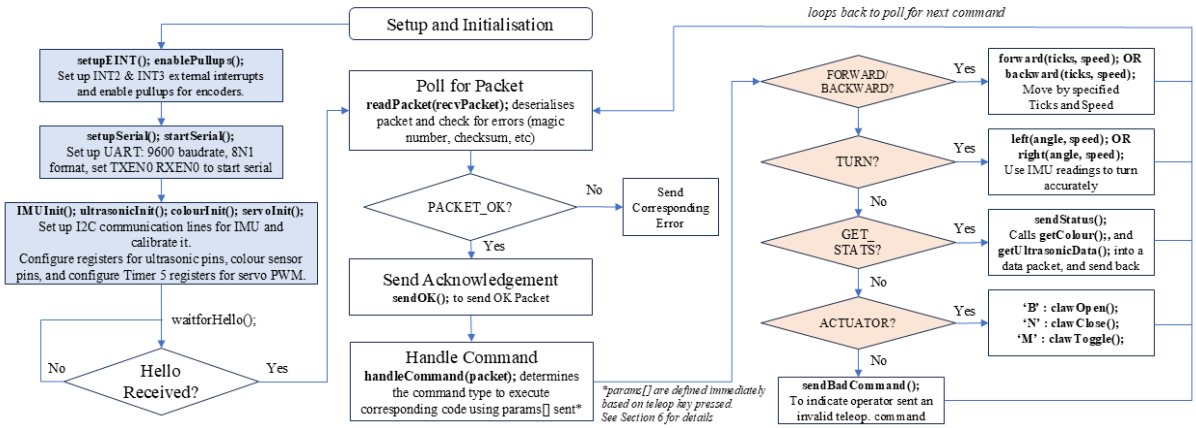


Figure 12: Flowchart of High-Level Algorithm on the Arduino

5.2 Communication Protocol

Communication between Arduino and RPi happens over the serial port via the UART protocol. The UART frame format used was 8N1 at a baud rate of 9600 bps. This is achieved with bare-metal code (refer to *Appendix C* for the code) by initialising UCSR0A to 0, UCSR0C to 0b110, UBRR0L to 103 and UBRR0H to 0 (for 9600 bps). Then, serial starts when we set UCSR0B to 0b11000 which sets the TXEN0 and RXEN0 bits, achieving serial comms via polling. In the main loop, Arduino receives serialised data from the RPi. Data is read by the Arduino by placing the bytes in UDR0 into a buffer when RXC0 in UCSR0A is set, then deserialising it into the `TPacket` struct in *Figure 13*. When Arduino returns to the RPi readings from the colour and ultrasonic sensor, it uses the same struct but fills the *params* array with the readings instead. Data is transmitted to the RPi by serialising the packet into a buffer, and then writing the buffer to UDR0 sequentially when UDRE0 in UCSR0A is cleared.

Format of Messages and Responces using struct TPacket

```
typedef struct
{
    char packetType;
    char command;
    char dummy[2]; // Padding to make up 4 bytes
    char data[MAX_STR_LEN]; // String data
    uint32_t params[16];
} TPacket;
```

Figure 13: TPacket struct used for Communications between RPi and Arduino

5.3 Movement Control and Peripheral Firmware Implementations

Motor speed is controlled via Pulse-Width Modulation (PWM), with each Output Compare Register linked to a specific motor using the AFMotor library. When the HW-130 motor shield is used with the Arduino Mega 2560, the library uses Timer1, Timer3, and Timer4 for PWM generation, each configured in Fast PWM mode. The timers are set to clear OCRnX on compare match, allowing precise control over motor speed by adjusting the duty cycles through the OCR values (OCR1A, OCR3C, OCR4A, OCR3A) individually. This low-level implementation is abstracted by the AFMotor library's methods `AF_DCMotor::setSpeed()` which sets the duty cycle, and `AF_DCMotor::run()`, which controls motor direction via latch logic. This simplified the motor control in our code.

5.3.1 Algorithm for Forward and Backwards

Instead of calculated distance, we used raw counts of encoder interrupt triggers, "ticks", in our `forward()` and `backward()` function to reduce rounding errors. Distance to travel is varied based on input ticks, which are mapped to teleoperating keys 'W' and 'S' for large, and 'I' and 'K' for fine movements. More details will be covered in *Section 6*.

5.3.2 Precise Turns using Adafruit BNO085 IMU Sensor

Turns are made by comparing `currentYaw` and `targetYaw = startYaw + turnAngle`. Alex spins clockwise/counterclockwise until the desired yaw value is achieved. The benefits of this

implementation are explained in *Section 4.4.1*. Our `getYaw()` function returns the real-time heading of Alex, and its full code is presented in *Appendix D*, with a snippet of it below:

```
if (bno.getSensorEvent(&sensorValue)) {
    if (sensorValue.sensorId == SH2_ROTATION_VECTOR) {
        // Convert quaternion to Yaw
        float qw = sensorValue.un.rotationVector.real;
        float qx = sensorValue.un.rotationVector.i;
        float qy = sensorValue.un.rotationVector.j;
        float qz = sensorValue.un.rotationVector.k;
        float siny_cosp = 2.0f * (qw * qz + qx * qy);
        float cosy_cosp = 1.0f - 2.0f * (qy * qy + qz * qz);
        float currYaw = atan2(siny_cosp, cosy_cosp) * 180.0f / PI;
        if (currYaw > 180) {
            currYaw -= 360.0;
        } else if (currYaw < -180) {
            currYaw += 360.0;
        }
        return currYaw;
```

Figure 14: Crux of the getYaw() Implementation to Obtain Real-Time Heading

Using the onboard SH-2 sensor fusion algorithm, the `SH2_ROTATION_VECTOR` encapsulates a fused quaternion derived from accelerometer, gyroscope, and magnetometer data. This quaternion is a continuous representation of Alex's 3D orientation in the form of rotation vectors. The quaternion is then converted into a yaw angle (Alex's rotation around the Z-axis and hence heading) in degrees using trigonometric relationships. We work within the range of $-180° < angle < 180°$ by wrapping accordingly. Values are hence centered around 0 within a 360 degrees range to make changes in direction more intuitive.

Common issues with IMU include drift in accelerometer and gyroscope data, and magnetic noise in magnetometer data. For drift, we simply call the SH-2 sensor fusion at 1000Hz, which consistently calibrates any drifts and inaccuracies in real-time. For magnetic noise, we used MotionCal, a visual magnetic field compensation tool that aligns magnetic field readings using scaling and offsets. (See *Appendix E* for the cool image!). This greatly mitigates inaccuracies due to drifts and noise, significantly improving the IMU's performance

5.3.3 Servo Algorithm that Powers the Actuator Mechanism

Two SG90 servos are driven using Timer 5 of the Arduino Mega (16-bit resolution and 50 Hz PWM signal) on Pins D46 (OC5A) and Pin D45 (OC5B). Setting OCR5A/B values controls the pulse width and hence position of the corresponding servo. (See *Appendix C* for the bare-metal code). We opted for this instead of the motor shield's servo header pins because they are connected to D10 and D9 which are 8-bit PWM, unable to generate the 50Hz PWM Signal that the servo motor operates at.

5.3.4 Ultrasonic Sensor Algorithm

The ultrasonic sensing algorithm emits a 10µs pulse via the TRIG pin and measures the time the ECHO pin stays HIGH using Timer2 configured with a prescaler of 64, yielding 4µs per tick on a 16MHz system. After timing, the timer count is converted to distance using the speed of sound. Implemented entirely in bare-metal programming (See *Appendix C* for the bare-metal code), the algorithm uses direct register access and avoids interrupts or blocking delays, enabling precise and efficient short-range distance measurement.

Section 6 - Software Design

6.1 High Level Algorithm on the Pi

1. Operator connects to the RPi via VNC.

2. Operator starts `AlexCameraStreamServer.py` to get the Picamera server running.
3. Operator can then access the Picamera through their browser by entering their RPi's IP address and the port opened for the Picamera. (Example: `192.168.239.102:8000`)
4. Operator starts the main teleoperation program, `alex_main.py`, which will attempt to connect to the Arduino and LIDAR connected to the RPi.
5. Once a display of the map shows, the operator can begin controlling the robot via commands sent through the terminal.

6.2 Enhanced Teleoperation and the "Astronaut Care Mode" using 'U' key

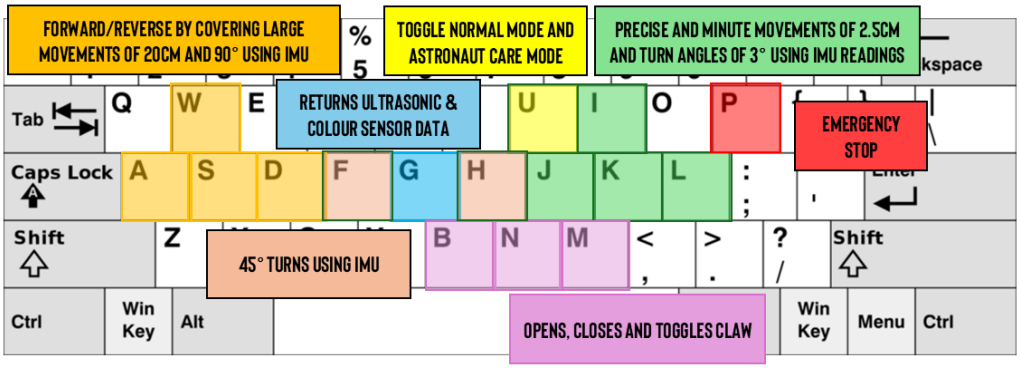


Figure 15: Teleoperation Keys

The keybinds above were used to control the robot. User inputs a command and sends it by pressing the Enter key. The template code given in the lab allows users to specify the distance / angle and speed of the movements. While this gave the user a great deal of freedom in movement, we decided it would be quicker to navigate with a set of predefined movements instead. (e.g. 'W' sends `parseParams([8, 45], 2, None)`while 'I' sends `parseParams([1, 45], 2, None)`). We have a unique "U" key which toggles between high and low speed, implemented using class and toggle (*Figure 16*). Grabbing the astronaut causes the motors to do more work to handle the increased weight and friction. While we accounted for this in our mechanical design in *Section 4.2*, increasing the motor power output increased the efficiency of our turns significantly. This software solution of "Astronaut Care Mode" also allows us to reuse the same movement keybinds since we chose not to have speed as an input.

Astronaut Mode Implementation Snippet

```python
class TurnPower:
    """
    TurnPower is a class used for storing turning power constants and toggling turning power between POWER_HIGH and POWER_LOW.
    """
    POWER_HIGH = 90
    POWER_LOW = 75
    def __init__(self):
        self.value = TurnPower.POWER_LOW
    def toggle(self):
        """
        Returns:
            True if turning power is POWER_HIGH and False otherwise.
        """
        self.value = (
            TurnPower.POWER_HIGH
            if self.value == TurnPower.POWER_LOW
            else TurnPower.POWER_LOW
        )
        return self.value == TurnPower.POWER_HIGH
turnPower = TurnPower()
```

```python
if command == "u":
    isHighTurnPower = turnPower.toggle()
    print("Turn Power: ", end="")
    print("High" if isHighTurnPower else "Low")
    return None
elif command == "a":
    commandType = TCommandType.COMMAND_TURN_LEFT
    params = parseParams([80, turnPower.value], 2, None)
    return (
        (packetType, commandType, params)
        if params is not None
        else print("Invalid Parameters")
    )
```

Figure 16: "Astronaut Care Mode" Class (left); Example of Implementation (right)

6.3 BreezySlam Optimisation

We used the provided BreezySlam with appropriate changes made to it for mapping of the environment. We first filtered out low quality LIDAR scans before publishing it to the

`LIDAR_SCAN_TOPIC`, as shown in *Figure 16*. We then changed some constants so that the map displays more accurately and frequently. (See *Appendix F* for the constants changed)

LIDAR Scan Filtering Implementations

```python
# relevant portion from lidarScanThread() found inside alex_lidar_scan_node.py
    try:
        scan_generator = lidar.start_scan_express(scan_mode)
        current_round = {"r": 0, "buff": [], "doScan": False}
        for count, scan in enumerate(scan_generator()):
            current_round, results = process_scan(
                (count, scan), scanState=current_round
            )
            if results and current_round["r"] > INITIAL_ROUNDS_IGNORED:
                angleData, distanceData, qualityData = results

                # Create a filter array using QUALITY_THRESHOLD
                goodQuality = np.array(qualityData) > QUALITY_THRESHOLD

                # Filter out low quality scans using the filter array
                angleData = np.array(angleData)[goodQuality]
                distanceData = np.array(distanceData)[goodQuality]
                qualityData = np.array(qualityData)[goodQuality]

                # Repack the arrays into the results tuple
                results = (angleData, distanceData, qualityData)

                publish(LIDAR_SCAN_TOPIC, results)

            if ctx.isExit():
                break
```

Figure 16: LIDAR Scan Filtering

6.4 TCS3200 Colour Sensing Algorithm

The TCS3200 sensor outputs frequency values corresponding to the intensity of reflected light through red, green, and blue filters. These frequency-domain signals were first normalized and reverse-mapped into the canonical RGB color space (0–255), since we noticed that the higher the values, the lower the actual intensity of the colour. Unlike past missions, our mission only consisted of two astronauts, in Green and in Red, which are both primary colours. There was no incentive in implementing robust algorithms or deployable ML pipelines like kNN classification. In fact, just with our shrouding in *Section 4.4.4*, Red values are already distinctly higher than the other two in the presence of red objects, and same for Green. Alex could therefore accurately discern between the two astronauts, regardless of ambient lighting.

6.5 Efficient Terminal Commands

We used bash alias to speed up the repetitive process of changing into the correct directories and enabling the python virtual environment.

```
alias activate="source ~/CG2111A/alex/env/bin/activate"
alias camera="python3 ~/CG2111A/Appendix-UsingYourCamera/AlexCameraStreamServer.py"
alias start="python3 ~/CG2111A/alex/labs/SlamLab/alex_main.py"
```

Figure 17: bashrc alias

Section 7 - Lessons Learnt & Conclusion

7.1.1 Lesson 1: BreezySlam and VNC are actually pretty decent in terms of speed

We were sceptical of the running speed of BreezySlam on the RPi and furthermore viewing it through VNC. Hence, we were contemplating whether or not to implement our own version of ROS's networking architecture. However, this meant we had to set up a TLS connection specifically for sending LIDAR data between the RPi and laptop. Figuring out how to parse the LIDAR data to send it from the RPi to the laptop proved to be a tedious task. In the end we settled for running BreezySlam on the RPi and viewing it via VNC. This saved us a lot of time which would otherwise be spent learning about TCP/IP and fixing more bugs.

7.1.2 Lesson 2: Always prepare contingency plans

While it did not happen to our group, we heard of other groups running into various problems during the 5 minutes set up given in the runs. For example, some groups' RPi failed to connect to their hotspot, Picamera failed or their VNC had high latency. Some of the problems we foresaw and had backup plans — RPi saved various hotspots in case one failed to connect — others however we did not plan for since we have never ran into them before.

7.2.1 Mistake 1: Spending too much time on trying to get ROS to work

We initially wanted to use ROS as we knew of its capabilities. In order to get ROS Noetic on RPi, we had to install an older version of RPi OS (Buster). This process took a while but we managed to get ROS Noetic working. However, we soon realised this version of RPi OS was incompatible with Picamera2, as it requires a newer version of RPi OS (Bullseye or later).

Since ROS Noetic did not work, we decided to try ROS2 instead. We installed ROS2 Humble using Docker on the Raspberry Pi and attempted to interface with it from our laptop via WSL. However, this approach introduced new complications—ROS2's discovery mechanism (Fast DDS) failed across WSL and Docker due to multicast and network interface issues. We spent considerable time adjusting firewall settings, configuring participant discovery domains, and troubleshooting Docker's host networking, but were unable to achieve stable multi-device communication.

In the end, we wasted a lot of time downloading and installing OS images, as well as trying to configure ROS2's networking and Docker environment, even configuring a new type of server and dual-booting on the laptop. Had we used the given python program earlier, we could have saved a lot of time which could be used for improving the robot. This taught us that even powerful tools like ROS require infrastructure compatibility, and overengineering can backfire when the foundational setup is unstable.

7.2.2 Mistake 2: Neglecting to practise our teleoperation procedures

We were too focused on the hardware and software side of things that we did not practise enough for the demo run. This led to us panicking and thus failing to hit the given objectives. Thankfully we still had time before the final run and we managed to nail down the protocols that should be followed during teleoperations. For example, to prevent confusion and time wasted making decisions, only one member is responsible for the entire movement of the robot. The others are in charge of sketching the map and providing information about what the camera sees.

References

1. *PackBot® 510*. PackBot® 510 | Teledyne FLIR. (n.d.). https://www.flir.com/products/packbot/?vertical=ugs&segment=uis

2. Paine, N., Mehling, J. S., Holley, J., Radford, N. A., Johnson, G., Fok, C. & Sentis, L. (2015). Actuator Control for the NASA-JSC Valkyrie Humanoid Robot: A Decoupled Dynamics Approach for Torque Control of Series Elastic Robots. *Journal of Field Robotics, 32*(3), 378-396. https://doi.org/10.1002/rob.21556

Afterword

From countless hours of testing to the thrill of Alex's final run, we found this project to be a very meaningful opportunity in learning, growing, and building something far beyond code and circuits. Here is a commemorative picture of us and Prof Henry Tan after our final run. Thank you for joining us on this journey.

Our Commemorative Picture during our Final Run

Appendix A - Images for Section 2

PackBot 510

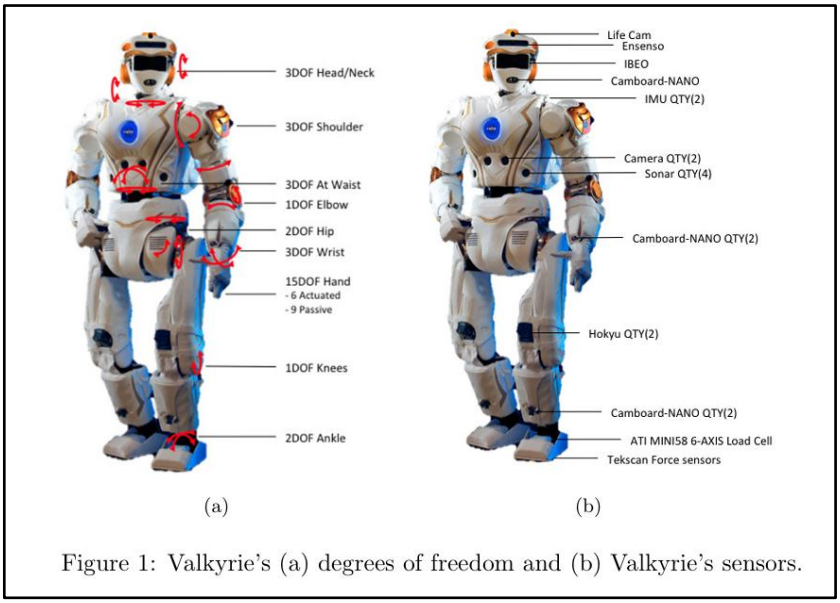
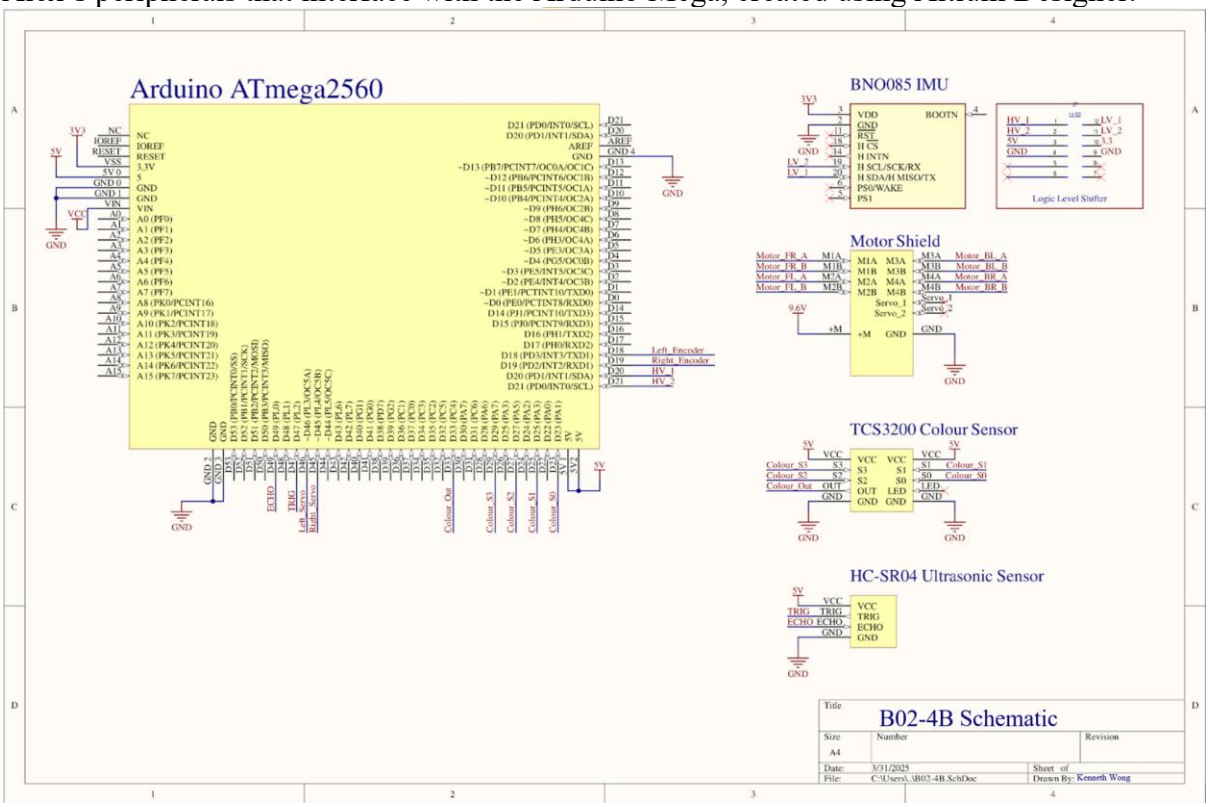


Figure 1: Valkyrie's (a) degrees of freedom and (b) Valkyrie's sensors.

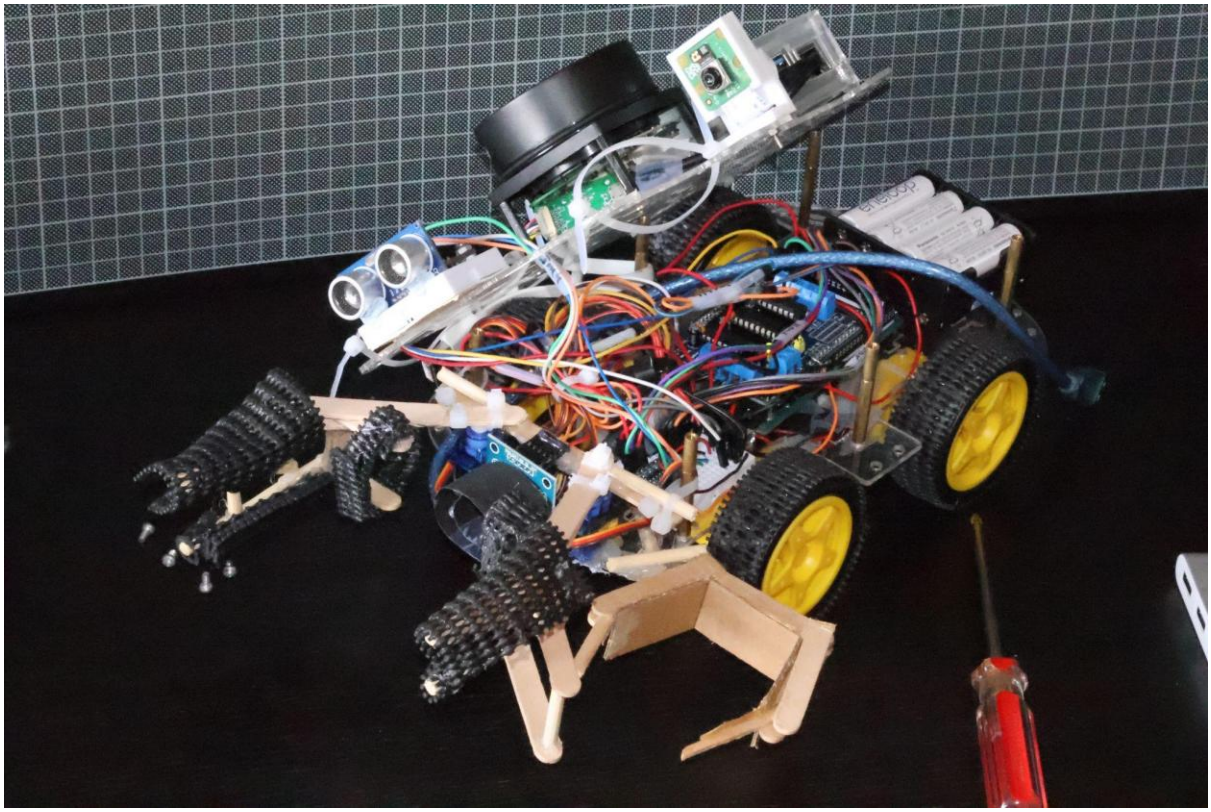
Valkyrie

Appendix B - Full Schematic of Alex's Electrical Design

The Arduino Mega acts as a real time hardware controller for motors, servo actuators, encoders and auxiliary sensors. The schematic below illustrates the electrical component-level view of Alex's peripherals that interface with the Arduino Mega, created using Altium Designer.



The picture below features our deconstructed Alex. Even with cable management, referring to the schematic saved us a lot of time when debugging.



Appendix C - Bare-Metal Codes for Servo, UART and Ultrasonic Sensor

This appendix contains snippets of our code in bare-metal for each of Alex's functionalities, in sequence as listed above

Bare Metal Code for Actuators (2x Servo)

```
#define SERVO_LEFT_PIN  PL3  // Arduino Pin D46, OC5A
#define SERVO_RIGHT_PIN  PL4  // Arduino Pin D45, OC5B

// for SG90 servos, 50 Hz PWM signal with a 20 ms period for servo control.
// Use Timer 5, left claw OC5A, right claw OC5B

bool clawClosed = false;

void servoInit() {
  // Set servo pins as output
  DDRL |= (1 << SERVO_LEFT_PIN) | (1 << SERVO_RIGHT_PIN);
  TCNT5 = 0;
  TCCR5A = 0;
  TCCR5B = 0;
  // Configure Timer 5 for 16 bit Phase Correct PWM (mode 10)
  // with ICR5 as TOP = fclk/(N*2*fpwm) = 16Mhz/(8*2*50Hz) = 20000, Prescaler of 8
  // Clear OCnA/B on match upcounting, set when downcounting
  TCCR5A |= (1 << COM5A1) | (1 << COM5B1) | (1 << WGM51);
  TCCR5B |= (1 << WGM53) | (1 << CS51);
  ICR5 = 20000;
  // Initially opened
  clawOpen();
}

/*
 * Map angle to OCR5 value by the pulse width
 * 0° = 1 ms pulse = 5% Duty Cycle
 * 90° = 1.5 ms pulse = 7.5% Duty Cycle
 * 180° = 2 ms pulse = 10% Duty Cycle
 */
void clawOpen() { // 'B' Teleop Command
  OCR5A = 1500;
  OCR5B = 300;
  clawClosed = false;
}
void clawClose() { // 'N' Teleop Command
  OCR5A = 1000;
  OCR5B = 700;
  clawClosed = true;
}
void clawToggle() { // 'M' Teleop Command
  if (clawClosed) {
    clawOpen();
  }else{
    clawClose();
  }
}
```

Bare-Metal Code for the two Servos used for the Actuator Mechanism

Bare Metal Code for UART Serial Communication

```c
void setupSerial() {
    UBRR0L = 103;
    UBRR0H = 0;

    UCSR0C = 0b00000110;
    UCSR0A = 0;
}

void startSerial() {
    UCSR0B = 0b00011000;
}

int readSerial(char *buffer) {
    int count = 0;
    while (UCSR0A & (1 << RXC0)) {
        buffer[count++] = UDR0;
    }
    return count;
}

void writeSerial(const char *buffer, int len) {
    for (int i = 0; i < len; i++) {
        while (!(UCSR0A & (1 << UDRE0)))
            ;
        UDR0 = buffer[i];
    }
}
```

Bare-Metal Code for the UART Serial Communication (9600 Baud Rate, 8N1 Format)

Bare Metal Code for Ultrasonic Sensor

```
#define TRIG_PIN PL2
#define ECHO_PIN PL0
#define TRIG_HIGH() (PORTL |= (1 << TRIG_PIN))
#define TRIG_LOW() (PORTL &= ~(1 << TRIG_PIN))
#define READ_ECHO() ((PINL >> ECHO_PIN) & 1)

void setup_ultrasonic_timer() {

    DDRD |= (1 << TRIG_PIN);
    DDRD &= ~(1 << ECHO_PIN);

    TCCR2A = 0x00;
    TCCR2B = (1 << CS22); // Prescaler = 64
    TCNT2 = 0;
}

uint32_t get_distance_cm()
    setup_ultrasonic_timer();
    TRIG_LOW();
    _delay_us(2);
    TRIG_HIGH();
    _delay_us(10);
    TRIG_LOW();

    while (!READ_ECHO());

    TCNT2 = 0;
    uint32_t ticks = 0;

    while (READ_ECHO()) {
        if (TCNT2 >= 255) {
            TCNT2 = 0;
            ticks += 255;}
}
    ticks += TCNT2;
    float time_us = ticks * 4;
    float distance_cm = (time_us * 0.0343) / 2;
    return (uint32_t)distance_cm;
}
```

Bare-Metal Code for Ultrasonic Sensor

Appendix D - Full IMU Code and Implementation

This code snippet shows the use of yaw angles in our turning algorithm. `currYaw` is a global variable that is obtained from our `getYaw()` function, which is defined in the full IMU code in the next page.

Using IMU yaw angles to turn left or right

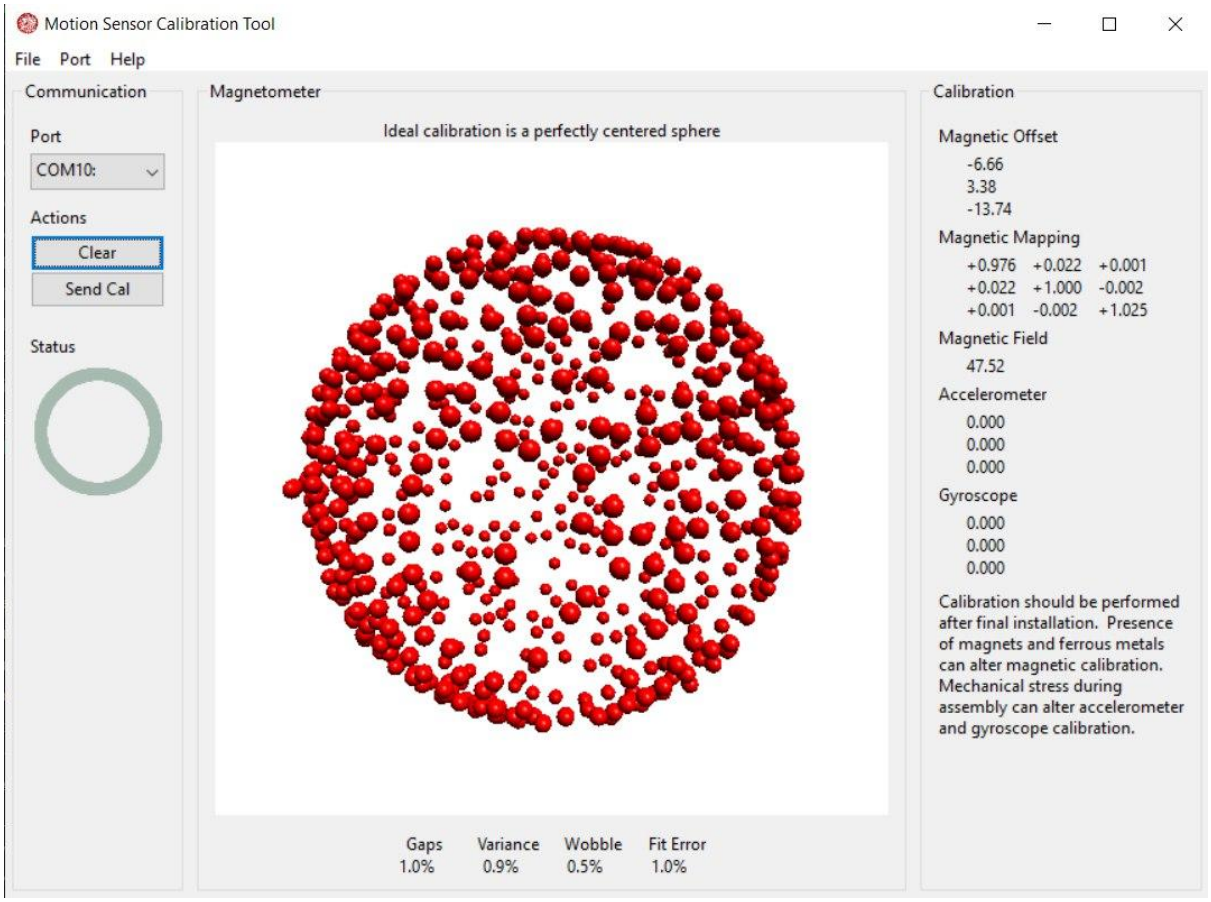
```
void left(float degree, float speed) {
    float startYaw = currYaw;
    float targetYaw = wrap(startYaw + degree);
    move(speed, CCW);
    while (true) {
        float diff = fabs(wrap(targetYaw - getYaw()));
        if (diff < 2.0)
            break;
        delay(5);
    }
    stop();
}

void right(float degree, float speed) {
    float startYaw = currYaw;
    float targetYaw = wrap(startYaw - degree);
    move(speed, CW);
    while (true) {
        float diff = fabs(wrap(targetYaw - getYaw()));
        if (diff < 2.0)
            break;
        delay(5);
    }
    stop();
}
```

BNO085 Implementation to Retrieve Real-time Yaw Angles (Alex's headings)

```
#include <Adafruit_BNO08x.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>
#include <math.h>

#define BNO08x_SDA 20
#define BNO08x_SCL 21
#define BNO08x_ADDRESS 0x4B // From I2C scan, I2C Default Address is 0x4A
Adafruit_BNO08x bno = Adafruit_BNO08x(-1); // -1 means no reset pin
float currYaw = 0.0;
float yawScaleFactor = 1.11273;
float yawOffSet = 20.07273;

void IMUInit() {
    // scanI2C();
    Wire.begin();
    if (!bno.begin_I2C(BNO08x_ADDRESS, &Wire)) {
        while (1);
    }
    // Set the sensor to use the SH-2 sensor fusion algorithm
    bno.enableReport(SH2_ROTATION_VECTOR); // yaw
}

/**
 * Get BNO08x Yaw readings from quaternion data using sh-2 sensor fusion
 */
sh2_SensorValue_t sensorValue;

float getSH2Yaw() {
    // Sensor fusion data structure
    if (bno.getSensorEvent(&sensorValue)) {
        if (sensorValue.sensorId == SH2_ROTATION_VECTOR) {
            // Convert quaternion to Yaw
            float qw = sensorValue.un.rotationVector.real;
            float qx = sensorValue.un.rotationVector.i;
            float qy = sensorValue.un.rotationVector.j;
            float qz = sensorValue.un.rotationVector.k;
            float siny_cosp = 2.0f * (qw * qz + qx * qy);
            float cosy_cosp = 1.0f - 2.0f * (qy * qy + qz * qz);
            float currYaw = atan2(siny_cosp, cosy_cosp) * 180.0f / PI;
            if (currYaw > 180) {
                currYaw -= 360.0;
            } else if (currYaw < -180) {
                currYaw += 360.0;
            }
            return currYaw;
        }
    }
    return currYaw;
}

float getYaw() {
    return yawScaleFactor * getSH2Yaw() + yawOffSet;
}
```

Appendix E - Magnetometer Calibration Visualisation

By tilting the IMU sensor in all directions, this tool maps the magnetic field readings using dots on the diagram. After rotating until a nice sphere is obtained, the IMU offsets are obtained. Using the offset values on the right, we scaled our yaw readings from the IMU in line 54 of the code in *Appendix D* as `yawScaleFactor` and `yawOffSet`.



Appendix F - Constants Configurations

Constants changed within `alex_slam_node.py`

```
# Map Constants
# Originally 5 but we increased it so the map updates more quickly
MAP_QUALITY = 20
MAP_SIZE_PIXELS = 500
MAP_SIZE_METERS = 7
MAP_SIZE_MILLIMETERS = MAP_SIZE_METERS * 1000
# Originally -90 but we changed it to 0 since our Lidar front
# is aligned with the robot
LIDAR_OFFSET_DEGREES = 0
```

Constants changed within `alex_display_node.py`

```
# SLAM constants
MAP_SIZE_PIXELS = 500
# Change map size to accurately reflect the actual size of the environment
MAP_SIZE_METERS = 7
MAP_SIZE_MILLIMETERS = MAP_SIZE_METERS * 1000
# Changed to reflect the actual dimensions of our robot
ROBOT_WIDTH_METERS = 0.27
ROBOT_HEIGHT_METERS = 0.36
```